The Tinker tool for graphical tactic development

Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-017-0452-7

TACAS 2016

The Tinker tool for graphical tactic development

Gudmund Grov[1] · **Yuhui Lin**[1]

Abstract *PSGraph* (Grov et al. in LPAR. Springer, Berlin, pp 324–339, 2013) is a graphical language to support the development and maintenance of proof tactics for interactive theorem provers. By using labelled hierarchical graphs this formalisation improves upon analysis and maintenance found in traditional tactic languages. Tool support for PSGraph is achieved by *Tinker* (Grov et al. in UITP 2014, ENTCS, vol 167. Open Publishing Association, London, pp 23–34, 2014; Lin et al. in Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 573–579, 2016): a theorem prover-independent system, which is connected to several different provers, with a graphical user interface including novel features to develop and debug proof tactics graphically. In this paper we provide a detailed and formal account of PSGraph and show how theorem prover independence is achieved by Tinker. We then show practical use of PSGraph and Tinker by developing several proof patterns using the language and tool.

Keywords Interactive theorem proving · Tactic languages · Development · Maintenance

This work has been supported by EPSRC Grants EP/J001058, EP/K503915, EP/H023852 and EP/H024204. The first author is supported by a SICSA Industrial Fellowship.

✉ Gudmund Grov
G.Grov@hw.ac.uk

Yuhui Lin
Y.Lin@hw.ac.uk

[1] Heriot-Watt University, Edinburgh, UK

1 Introduction

Most interactive theorem provers provide users with a tactic language in which they can encode common proof strategies in order to reduce user interaction. To encode proof strategies, these languages typically provide: a set of functions, called *tactics*, which reduces goals into smaller and simpler sub-goals; and a set of combinators, called *tacticals*, which combines tactics in different ways. Tacticals typically explore higher-order features in the tactic languages, which enable users to write short and concise strategies. However, they can easily become hard to understand and even harder to analyse and debug. When a tactic is used to explain a proof strategy to a stakeholder, or maintained and further developed by others than the original developer, then it is crucial to have a representation that is intuitive[1]. For example, consider the following proof strategy, called `simple_quantifier_tac`, implemented using the tactic language of the ProofPower system [4]:

```
REPEAT ((REPEAT strip_∧) THEN
        (TRY (all_∃_uncurry ORELSE
              redundant_simple_∃ ORELSE
              simple_∃_equation ORELSE
              simple_∃_∧)) THEN
        (TRY (all_∀_uncurry ORELSE
              redundant_simple_∀ ORELSE
              simple_∀_∧ ORELSE
              simple_∀_tac)))
```

To fully grasp this strategy one needs to understand the detailed semantics of the various tacticals, such as `REPEAT` and `ORELSE`. For example, when does `REPEAT` terminate? Does it require the given tactic to run at least once? Or will

[1] See, e.g. [29], which addresses industrial use of PSGraph.

Published online: 17 March 2017

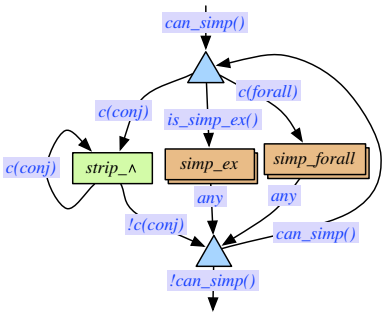


Fig. 1 A PSGraph example

it succeed if it cannot run to begin with? We have found that many mistakes are due to misunderstanding of such corner cases [28]. There is also the issue of debugging. How can one find the cause of a failure of the tactic? Or, possibly worse, the cause of a success but with an unexpected result. Such bugs are very hard to locate. Debugging is made even harder by "defensive programming" through the `TRY` tactical, which either applies a tactic or does nothing, as it is hard to see the overall strategy. The most common solution to find bugs is to manually break the tactic apart into sub-tactics and use, e.g. `writeln` statements to see the proof state at various points during evaluation.

To overcome these issues we have developed *PSGraph*, a graphical proof strategy language [16], where tacticals are replaced by directed, typed and hierarchical graphs. The boxes contain tactics provided by the underlying prover or nested graphs. These are connected by labelled wires. The labels are called *goal types* and are predicates describing expected properties of a goal[2]. Each goal becomes a special *goal node* on the graph. These are passed between tactics over the wires. To pass a goal over a wire, it has to satisfy the wire's goal type.

To illustrate, Fig. 1 shows a PSGraph representation of the above tactic with two components abstracted out:

```
REPEAT ((REPEAT strip_∧) THEN
          (TRY simp_ex) THEN
          (TRY simp_forall))
```

The first thing to notice is that `simple_quantifier_tac` repeatedly (`REPEAT`) applies three components in sequence (using `THEN`). It is unlikely that the developer had such a sequential ordering in mind; it is more likely a by-product where the tactic language enforces an order (combined with "defensive" programming using `TRY`).

The corresponding PSGraph draws them in parallel, which is likely to be closer to the high-level strategy that the developer had in mind. The goal types on the wires are used to explicitly direct the goals to the suitable tactic, while also being explicit about issues such as termination conditions for loops. The advantage is that a user can read the overall strategy directly from the graph, with the justification for the choices (via goal types), without detailed knowledge of the semantics of the tacticals used[3]. For this particular example, if we know that *strip_∧* strips conjunctions and *simp_ex* and *simp_forall* eliminate existential and universal quantifiers, then one can see directly from the graph what the strategy does. We will return to the details of this strategy in the next section, when describing the PSGraph language.

PSGraph is supported by a tool called Tinker [17], which is connected to several theorem provers. In [30], we described a new version of Tinker with novel GUI features for developing and debugging PSGraphs. This paper is an extended version of [30], with the following additional contributions:

- We give a formal account of PSGraph, including a formal operational evaluation semantics.
- We encode several proof strategies in PSGraph to illustrate usability.
- We provide details of how theorem prover independence is achieved by Tinker.

In Sect. 2 we describe the PSGraph language, while in Sect. 3 we show how it is used to prove conjectures, including the formal evaluation semantics. In Sect. 4 we provide details of the Tinker tool, while Sect. 5 describes how to develop and debug proof strategies using the Tinker GUI. In Sect. 6 we show several proof strategies encoded in PSGraph, before we discuss related work (Sect. 7) and conclude (Sect. 8).

2 The PSGraph language

An important part of PSGraph is the goal types. We first give an exposition of goal types before discussing the PSGraph language in general.

2.1 Goal types

A *goal type* is a predicate over a goal. Each wire of a PSGraph is labelled by a goal type. The simplest goal types are predicates on the goal terms: e.g. the PSGraph of Fig. 1 has a wire with the goal type *c(conj)*, which states that the conclusion of the goal is a conjunction (i.e. of the shape "_ ∧ _").

The language used to express goal types was introduced in [28]. It combines *atomic goal types*, which are relations provided by the provers, with a Prolog-based language that combines them. Following Prolog convention, constants start with lower case (e.g. *x*, *y*, *xs*), and goal type variables start

[2] We use *goal* for both the top-level goal and any sub-goal.

[3] The brain also finds diagrammatic representations more natural to understand for such "process systems" [24].

with upper case (e.g. X,Y, Xs). Two common constants are *concl*, which is the goal conclusion, and *hyps*, the list of hypotheses. Terms used by the prover can also be encoded in the goal types. In addition to goal type variables, there is another notion of variables, which are prefixed by "?" (e.g. $?x$, $?y$, $?xs$). They are treated as constants (or terms) in goal types. We will return to them below.

A *goal type* is a non-empty list of (possibly negated) relations or defined goal types, written as

$$c_1(a_{01}, \ldots, a_{n1}), \ldots, c_m(a_{0m}, \ldots, a_{km}).$$

To declutter graphs and increase expressiveness, we can introduce new definitions through a *goal type schema*. It has the form

$$h(X_1, \ldots, X_n) \leftarrow Body$$

where *Body* is a list of relations. To illustrate, consider the goal types of Fig. 1. Here, *c(conj)* and *c(forall)* are defined using *c(X)*:

$$c(X) \leftarrow top\_symbol(concl, X).$$

This definition uses the atomic goal type

$$top\_symbol(S, T)$$

which states that symbol S is the top symbol of term T. The *can_simp*() goal type is defined in terms of the other goal types:

$$can\_simp() \leftarrow c(conj).$$
$$can\_simp() \leftarrow c(forall).$$
$$can\_simp() \leftarrow is\_simp\_ex().$$

Here, the definitions of *is_simp_ex*() are omitted. Each clause should be seen as a disjunction, meaning the goal type is satisfied if either of them are satisfied. Finally, in order to negate $c_1(a_{01}, \ldots, a_{n1})$ one writes $!c_1(a_{01}, \ldots, a_{n1})$. This is illustrated by !*can_simp*() in Fig. 1. We will see more advanced goal types in Sect. 6.

2.2 Graphical proof strategies

Figure 2 shows the types of boxes that a PSGraph may contain: an *atomic tactic* is a box that is labelled by a tactic of the underlying theorem prover or an environment tactic (discussed below); a *graph tactic* is a box labelled by a named nested graph, which is used to handle modularity; an *identity tactic* does not change the goal and is used to merge and split wires; a *goal* node contains an open goal to be proven. This

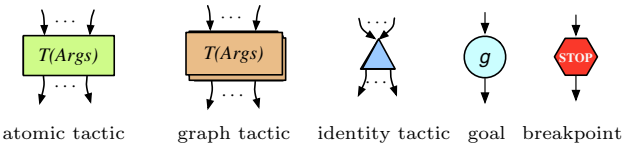


Fig. 2 Different types of *boxes* used in a PSGraph

can only be added or changed by tactic application; *breakpoints* are used to control evaluation, which we return to in Sect. 5.

At a given time during the proof of a conjecture using PSGraph, there are one or more open goals that has to be proven on the graph. A theorem prover will keep track of these in a *proof state*. A tactic application will be applied to one goal of the proof state and will replace this goal with new generated goals, if any.

A *goal* node in a graph will be labelled by a goal of the proof state. In addition, it contains an *environment*. The environment is a map from a named variable, which is prefixed by "?", to a term, another named entity such as a named lemma or a list of them. It is used for communication between tactics, and to express global constraints in the goal types.

An *identity box* does not change the goal it evaluates. It is used to split and merge the paths a goal can take. Figure 1 illustrates both these features, where an identity box is used together with the goal type to send a goal to the correct destination (split) and then to merge the outputs again.

Modularity and *scalability* are achieved by nesting subgraphs inside *graph tactics*. PSGraph keeps track of all named child graphs, and a graph tactic box in a graph is labelled by such a name together with arguments. The arguments have to be variables and are used to ensure local scoping. For example, consider a graph tactic

$$h(?x, ?y).$$

This will introduce a local scope in the graph nested by h, such that only $?x$ and $?y$ will be kept from the environment of a goal node entering this nested graph. On exit, only $?x$ and $?y$ will be changed from the environment of the goal that entered the nested graph.

Figure 1 contains two graph tactics, *simp_ex* and *simp_forall*. Figure 3 shows the graph nested by *simp_ex*, which corresponds to the (sub-)tactic:

```
all_∃_uncurry ORELSE
redundant_simple_∃ ORELSE
simple_∃_equation ORELSE
simple_∃_∧
```

The individual tactics work as follows: `all_∃_uncurry` changes paired quantifiers to uncurried versions; `redundant_simple_∃` removes the quantified variables

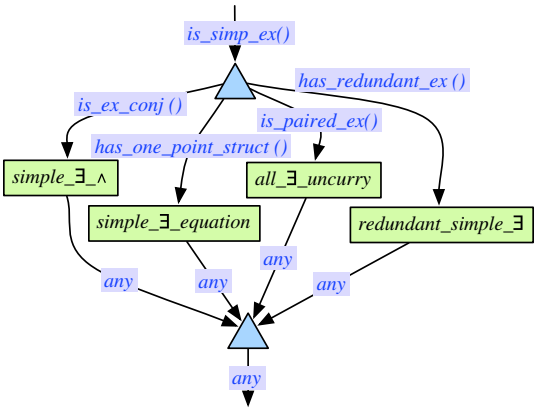


Fig. 3 A PSGraph example: *simp_ex* graph tactic

if they are not used in the body; `simple_∃_∧` distribute quantifiers over ∧; `simple_∃_equation` simplifies goals with the one point rule [40] (see Sect. 6). These are attempted in the given order until one of the tactics succeeds. As *simp_ex* is prefixed by `TRY`, the tactic will do nothing if all the tactics fail. *simp_forall* is omitted as its encoding is similar.

In the PSGraph version of this code, shown in Fig. 3, the order does not matter: a goal will be applied to the tactic where the incoming goal type succeeds. These are therefore drawn in parallel. If there are multiple goal types that succeed then all of them are tried. For speed and maintainability, it is important to have as non-deterministic strategies as possible. One thus needs to take care when developing goal types. Note that we ignore the overall `TRY` tactical as the goal type on the input wire should ensure that a tactic is applicable.

Figure 3 uses the same tactics as the code. From PSGraph's perspective, these tactics are the atoms, as they becomes "black boxes" that cannot be split further. We therefore call them *atomic tactics*. A box with an atomic tactic is labelled by the tactic name and optional arguments which we return to below. Figure 1 also contains a single tactic called *strip_∧*; it will break a conjunction into a new goal for each of the conjuncts. As the feedback-loop label shows, this is applied as long as the conclusion is a conjunction (goal type *c(conj)*). The overall strategy is applied as long as one tactic is applicable, identified by the *can_simp()* goal type.

The other class of atomic tactics is the *environment* tactics. These are tactics that only update the environment in a goal node. This is the only way to change an environment. An environment tactic has a name that starts with "*ENV_*". To illustrate,

ENV_bind(t, ?v)

will bind variable ?v to t in the environment. Once bound, ?v can be used in a goal type:

top_symbol(∃, ?v)

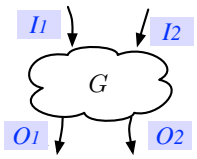
It can also be used by an atomic tactic. For example,

rule_tac(*exI*, ?v).

will instantiate a witness with the value of ?v.

3 Proving with PSGraph

A PSGraph may be *open* [14], in that a wire may not have a source or not have a destination. A wire without a source is an *input* for the graph, while a wire without a destination is an *output* of the graph. To illustrate, consider a graph G with two wires without a source, labelled by goal types I_1 and I_2, and two wires without a destination, labelled by goal types O_1 and O_2:



In order to apply such a PSGraph to (possibly partially) prove a conjecture, the conjecture has to be initialised first in the *proof state* of the underlying theorem prover. The goal then has to be wrapped in a goal node:

Definition 1 (*Goal*) A *goal node*, represented by the type *Goal*, contains a goal *name* of the actual goal in the proof state and an environment *env*.

A goal node can then be added to one of the input wires. In order to add the goal to an input, it must satisfy the goal type. That is, if it succeeds for I_1 then it can be added to this wire. If it succeeds for both I_1 and I_2 then both of them are tried separately. Once added, this goal, or any goals generated from subsequent tactic applications, will then "flow" through the graph until all the goals are on the output wires, i.e. O_1 or O_2. At this point evaluation has *terminated*:

Definition 2 (*Termination (normal mode)*) A graph has *terminated*, if for all goals g of the graph, the destination of the output wire for g is either the graph output or another goal.

It follows by induction over the number of goals present that all goals must be on the output wires on termination.

We can reduce the discussion of evaluation to a single step over one of the boxes of the graph. A full graph is evaluated by repeating such steps until termination is reached. This process uses and updates two components: the PSGraph *PG*, which keeps track of the state of the strategy; and a proof state *P*, keeping track of the goals and the necessary bookkeeping required by the theorem prover. The proof state is handled

$$\frac{\begin{array}{ccc} PG \vdash h_{10} : H_1 & \cdots & PG \vdash h_{1m} : H_1 \\ \vdots & & \\ PG \vdash h_{n0} : H_n & \cdots & PG \vdash h_{nk} : H_n \end{array} \quad \langle PG, P, T(Args), g\rangle \Downarrow_T \langle P', hs\rangle \quad hs = [h_{10}, \ldots, h_{n0}, \ldots, h_{n0}, \ldots, h_{nk}]}{}$$

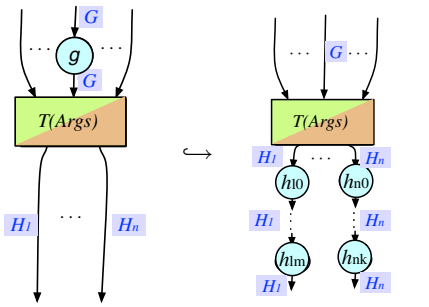


Fig. 4 Schematic rewrite rule for atomic/graph tactics

by the underlying prover and thus will vary between provers as detailed in § 4.

At the graph level, evaluation works by rewriting. A rewrite rule is written as $L \hookrightarrow R$, which, when applied, will replace a sub-graph L with R. The rewriting uses rich *pattern graphs* [21,22], which is used to express repetition using ellipses ($\cdots$). A formal account of pattern graphs can be found [21,22] and is beyond the scope of this paper. To increase readability, we omit some of the goal types where they are not used, but note that each wire will have a label even if not shown.

Figure 4 illustrates the rewrite rule schema to evaluate an *atomic* or *graph* tactic. Note that when there are no arguments, then we can write T instead of $T()$. The rule has several conditions, written above the line. Conditions of the form

$$PG \vdash g : G$$

express that a goal g satisfies goal type G, given the PSGraph PG. The other relation

$$\langle PG, P, T(Args), g\rangle \Downarrow_T \langle P', gs\rangle$$

is the semantics to evaluate a goal by tactic T, producing a new proof state P' and new goals gs. A formal account of this relation is given in § 3.1.

The steps for an *atomic tactic* are:

1. Consume g from the graph.
2. Apply $T(Args)$ to g to obtain a set of results (lists of goals paired with a proof state) from the application of tactic T with the given arguments $Args$.
3. Add all valid combination of the resulting goals to the output wires. These are combined with the new proof state.

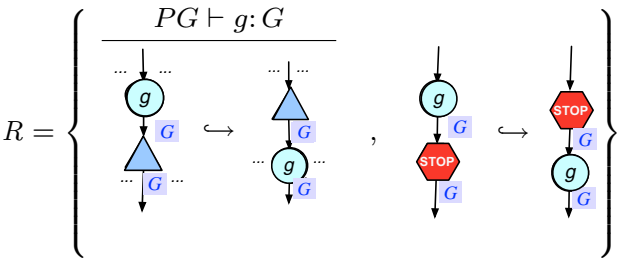


Fig. 5 Evaluation of identity tactics and breakpoints

A special case of this tactic is when T only makes changes to the environment; in this case, the proof state will remain unchanged.

For a *graph tactic*, the steps are:

1. Look up the graph G that T refers to
2. Consume g from the graph, and create g' where the environment is constrained to the variables in $Args$.
3. Add g' to all the input wires of G where g' satisfies the goal type (one branch for each), and evaluate until termination. The resulting goals should now be on the output wires.
4. Add all valid combinations of the resulting goals to the output wires of T. When adding these, the environment should be replaced by the environment of g, updated by any changes to variables in $Args$.

In addition to atomic and graph tactics, a graph can also contain *identity tactics* and *breakpoints*. These have no side effects on the proof state and are shown as the rewrite ruleset R in Fig. 5. A goal g is simply "moved over" an identity box or breakpoint, as long as the goal type of the target wire is satisfied. A breakpoint is only allowed to have one input and one output wire, labelled by the same goal type. The final type of node is a goal node, and it is not allowed to "swap" two adjacent goal nodes, albeit this may be added in the future to implement heuristic-based evaluation strategies.

A PSGraph is evaluated (to completion) by applying the rewrite rules outlined above until none are applicable. If the termination condition holds at that point, then it has *successfully* evaluated; if not, evaluation has *failed*. In § 5 we will discuss a "debugging" mode with a slightly different semantics.

3.1 A formal account of evaluation

To give a formal account of PSGraph, we use a VDM-based mathematical notation [5]. In particular, note that: P-**set** is the set of all subsets of P; T^* is a sequence of type T, where [] is the empty sequence and **cons**(x, xs) adds element x to

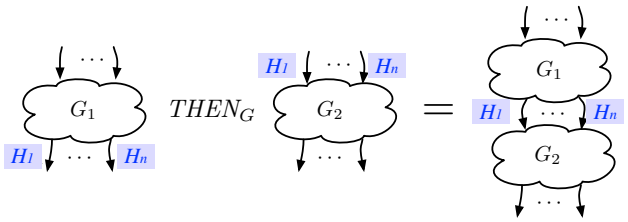
sequence xs; **elems** S is the (set of) elements of sequence S; **dom** R is the domain of relation R; domain subtraction is represented by $\lhd$, while its dual, domain restriction is written $\rhd$.

We do not provide the operational semantics for a goal type ($PG \vdash g : G$) as this is given in [28], where the goal type language was introduced.

We use the type *Graph* to describe an actual graph of a PSGraph, which is an instance of a *string diagram* [14]. We write $G[L \hookrightarrow R]$ for the application of the rewrite rule $L \hookrightarrow R$ to graph G. This will return a set of new graphs. We write $G_1 \hookrightarrow_R G_2$ for applying a rule in the ruleset R to rewrite G_1 into G_2. Rewriting is achieved by *matching* L with the graphs we are rewriting. Then this matching sub-graph is removed from the graph and replaced by R.

We use two combinators to compose graphs:

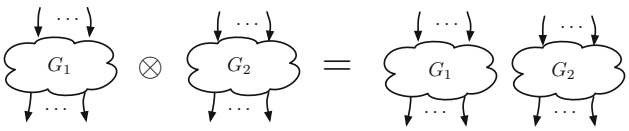
Definition 3 (*THEN*$_G$ *combinator*) G_1 *THEN*$_G$ G_2 connects all the outputs of G_1 to all inputs of G_2 of the same type. Diagrammatically this can be seen as follows:



$THEN_G$ is only defined when all outputs/inputs of G_1/ G_2 are connected.

Horizontal composition is a result of putting the two graphs next to each other:

Definition 4 ($\otimes$ *combinator*) $G_1 \otimes G_2$ is the horizontal combinator which puts two graphs side by side:



Details of the semantics underpinning both the rewriting and composition are beyond the scope of this paper, and we refer to [14] for details.

We can now define a *PSGraph*:

Definition 5 (*PSGraph*) A PSGraph PG contains the following fields

$graph(PG)$: *Graph*
$current(PG)$: *Graph*
$children(PG)$: *Name* $\xrightarrow{m}$ *Graph*

graph is the main (top-level) graph, *current* is the focus for evaluation, while *children* is a map from a name to a graph and contains the child nested by the named graph tactic. $PG[f := e]$ is the PSGraph PG with field f replaced by e.

The evaluation relation $\Downarrow$ is a relation over two pairs of a "before" PSGraph PG and proof state P, and an "after" PSGraph PG' and proof state P':

$$\langle PG, P\rangle \Downarrow \langle PG', P'\rangle$$

An alternative is to also provide a goal g (which exists in the proof state). This case is achieved by the $\Downarrow_G$ relation, which is defined in terms of the $\Downarrow_G^*$ relation:

$$\frac{\langle PG, P, graph(PG), g\rangle \Downarrow_G^* \langle PG', P'\rangle}{\langle PG, P, g\rangle \Downarrow_G \langle PG', P'\rangle}$$

Figure 6 gives the evaluation semantics. The $\Downarrow_G^*$ relation (EvalGraph) evaluates a goal g on a given graph G by first adding the goal to the input (AddInput) and then sets the *current* field to this graph. It then evaluates the graph until termination (Definition 2) using $\Downarrow^*$: the reflexive transitive closure of $\Downarrow$. EvalGoal ($\Downarrow_G^*$) is a simpler version of EvalGraph, which uses the current graph instead of a given graph, and is used to initialise a PSGraph with a goal. It is straightforward to extend the semantics to support multiple input goals as well.

The $\Downarrow$-relation has two cases: Rewrite represents the steps handled by rewriting only with the ruleset R, given in Fig. 5. Step relates to a step of an atomic or a graph tactic, as illustrated by the rule schema in Fig. 4. The generation of the left-hand side L of the rewrite is handled by matching with the *current* graph of the PSGraph. This will produce all possible matches, i.e. all possible goal nodes that can be evaluated. This is denoted by *lhs*:

$$lhs \in Graph \xrightarrow{m} Graph\text{-}\mathbf{set}.$$

The right-hand side R is generated by the $\Downarrow_R$ relation, and the current graph is updated by rewriting L to R. As all goal nodes are unique, it is only one way to apply this rule (see Claim 3).

$\Downarrow_R$ is captured by Right, and R is generated by first removing the goal g by rewriting and then applying the tactic $T(Args)$ using the $\Downarrow_T$ relation to get the new goals. From the output goal types, a valid *goal type partition* is created from the output goals:

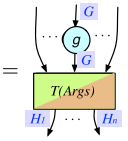
Definition 6 (*Ordered partition* [16]) For a sequence L, we say a sequence of sequences L' is an *ordered partition* of L if all the sequences are distinct, L' contains the same elements as L and each $l \in$ **elems** L' is obtained by deleting zero or more elements of L (i.e. the order of L is preserved).

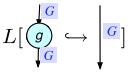
$$\textsc{EvalGraph}\ \frac{\langle PG, G, gs\rangle \Downarrow_I G' \quad \langle PG', P\rangle \Downarrow^* \langle PG'', P'\rangle \quad G' = current(PG'') \quad Terminated(G')}{\langle PG, P, G, gs\rangle \Downarrow^*_{GG} \langle P', G'\rangle}$$

$$\textsc{EvalGoal}\ \frac{\langle PG, current(PG), empty\_goal[name := g]\rangle \Downarrow_I G' \quad \langle PG[current := G'], P\rangle \Downarrow^* \langle PG'', P'\rangle \quad G' = current(PG'') \quad Terminated(G')}{\langle PG, P, g\rangle \Downarrow^*_G \langle P', G'\rangle}$$

$$\textsc{Rewrite}\ \frac{PG \vdash current(PG) \hookrightarrow_R cur' \quad PG' = PG[current := cur']}{\langle PG, P\rangle \Downarrow \langle PG', P\rangle}$$

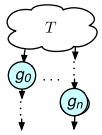
$$\textsc{Step}\ \frac{L \in lhs\Big(current(PG)\Big) \quad \langle PG, P, L\rangle \Downarrow_R \langle P', R\rangle \quad G' \in current(psgraph)[L \hookrightarrow R] \quad PG' = PG[current := G']}{\langle PG, P\rangle \Downarrow \langle PG', P'\rangle}$$

$L =$ 

$R_1 = L[$ 

$]$

$$\textsc{Right}\ \frac{\langle PG, P, T(Args), g\rangle \Downarrow_T \langle P', gs\rangle \quad [p_1, \ldots, p_n] \in part([H_1, \ldots, H_n], gs) \quad R_2 = gnds(p_1, H_1) \otimes \cdots \otimes gnds(p_1, H_1) \quad R = R_1\ THEN_G\ R_2}{\langle PG, P, L\rangle \Downarrow_R \langle P', R\rangle}$$

$$\textsc{Graph}\ \frac{T \in \mathbf{dom}\ children(PG) \quad e_r = \mathbf{elems}\ Args \lhd g(env) \quad g_a = g[env := \mathbf{elems}\ Args \lhd g(env)] \quad \langle PG, P, children(PG)(T), g_a\rangle \Downarrow^*_{GG} \langle P', G\rangle \quad G = \text{  } \quad newg(goal) := goal[env := e_r \cup (\mathbf{elems}\ Args \lhd goal(env)] \quad gs = [newg(g_0), \ldots, newg(g_n)]}{\langle PG, P, T(Args), g\rangle \Downarrow_T \langle P', gs\rangle}$$

$$\textsc{Atomic}\ \frac{T \notin Env \quad \mathbf{elems}\ Args \cap Vars \subseteq \mathbf{dom}\ env(g) \quad (P', [g_1, \ldots, g_n]) \in eval\Big(T, [\![Args]\!]_{env(g)}, P, name(g)\Big) \quad gs = \Big[g[name := g_1], \ldots, g[name := g_n]\Big]}{\langle PG, P, T(Args), g\rangle \Downarrow_T \langle P', gs\rangle}$$

$$\textsc{Env}\ \frac{T \in Env \quad e' \in eval\_env\Big(T, Args, env(g)\Big) \quad gs = \Big[g[env := e']\Big]}{\langle PG, P, T(Args), g\rangle \Downarrow_T \langle P, gs\rangle}$$

$$\textsc{AddInput}\ \frac{[GT_1, \ldots, GT_i, \ldots, GT_n] = inputs(G) \quad PG \vdash g : GT_i \quad IG = gnds([], GT_1) \otimes \cdots \otimes gnds([g], GT_i) \otimes \cdots \otimes gnds([], GT_n) \quad G' = IG\ THEN_G\ G}{\langle PG, G, g\rangle \Downarrow_I G'}$$

Fig. 6 Evaluation semantics of PSGraph

Definition 7 (*Goal type partition* [16]) For a PSGraph PG, a sequence of goal types $[G_1, \ldots, G_n]$ and a sequence of goals $[g_1, \ldots, g_m]$, a *goal type partition* is an ordered partition:

$$P = \Big[[g_i, g_{i'}, \ldots], [g_j, g_{j'}, \ldots], \ldots\Big]$$

such that for each g'_k in the k-th list in P it is the case that

$$PG \vdash g'_k : G_k.$$

In general, there may be more than one way to partition a list of goals. Given a list of goal types and a list of goals, let *part* be the set of all possible ways to goal type partition these goals:

$$part \in Goal\ type^* \times Goal^* \to P\textbf{-set}$$

The set of partitions is empty precisely when there is a goal in L that is not of type G_k for any k.

Example 1 To illustrate, consider a sequence of goals $[g_1, g_2, g_3]$ and a sequence of goal types $[G_a, G_b]$, such that

$$PG \vdash g_1 : G_a \quad PG \nvdash g_1 : G_b$$
$$PG \nvdash g_2 : G_a \quad PG \vdash g_2 : G_b$$
$$PG \vdash g_3 : G_a \quad PG \vdash g_3 : G_b$$

For this example, $part([G_a, G_b], [g_1, g_2, g_3])$ returns a set of two partitions

$$\Big\{\big[[g_1, g_3], [g_2]\big], \big[[g_1], [g_2, g_3]\big]\Big\}$$

since g_1 only satisfies G_b, g_2 only satisfies G_b, and g_3 satisfies both G_a and G_b.

Each partition is turned into a graph by *gnds*:

$$gnds([], G) := \text{}$$
$$gnds(\mathbf{cons}(g, gs), G) := gnds(gs, G)\ THEN_G\ \text{  }$$

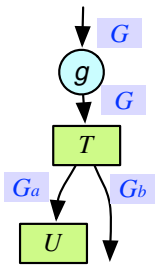
These are then horizontally composed and then sequentially composed with the left-hand side (with the input goal removed). Evaluation of an atomic box (ATOMIC) is achieved ($\Downarrow_T$ relation) by a function *eval* that executes the function. This has to be provided by the underlying prover. We return to how this is achieved in § 4, when discussing the Tinker tool.

Let *Vars* be the set of all variables (which are prefixed by "?"). All variables in *Args* must be found in the environment of the goal. To apply a tactic with *eval*, the variables are first instantiated by the values in the environment:

$$\begin{aligned}[\![\,[\,]\,]\!]_E &:= [\,]\\ [\![\mathbf{cons}(?v, xs)]\!]_E &:= \mathbf{cons}(E(?v), [\![xs]\!]_E)\\ [\![\mathbf{cons}(x, xs)]\!]_E &:= \mathbf{cons}(x, [\![xs]\!]_E)\end{aligned}$$

The tactic is then only given the named goal, and *eval* returns a set of proof states paired with a set of goal names. Each goal name is turned into a goal node with the same environment as the goal node it was generated from.

Example 2 To ilustrate evaluation of an atomic tactic, consider a pair of a PSGraph PG and proof state P, such that the *current* graph of PG is:



To evaluate goal g using tactic T, the new proof states and sub-goals are first computed by

$$eval\big(T, [\![]\!]_{env(g)}, P, name(g)\big).$$

Assume that this tactic application returns the following set:

$$\Big\{\big(P_1, [g_1, g_2, g_3]\big), \big(P_2, [g_4]\big), \big(P_3, [g_5]\big)\Big\}$$

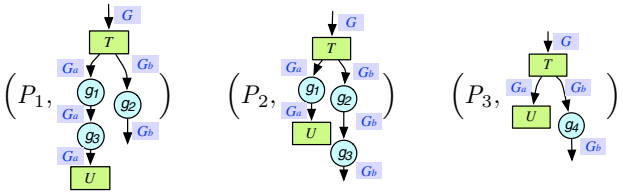
This means that there are three possible results when applying T to g. Further assume that g_1, g_2 and g_3 have the properties from example 1 and that

$$\begin{aligned}PG &\nvdash g_4 : G_a \qquad PG \vdash g_4 : G_b\\ PG &\nvdash g_5 : G_a \qquad PG \nvdash g_5 : G_b.\end{aligned}$$

For each element of the set, the goal type partition for the new sub-goals is first computed. This is computed with respect to the output wires of T, which are labelled by goal types G_a and G_b, forming the sequence $[G_a, G_b]$:

- For the first element, the goal type partition is given in example 1.
- For the second element, the singleton set $\Big\{\big[[\,], [g_4]\big]\Big\}$ is returned.
- For the third and final element, the empty set {} is returned. This means that a goal cannot be added to any of the output wires. As a result, this element is discarded.

One branch of the search space is then generated for each partition. For each of these, the current graph is updated by consuming the input goal g and adding the goals to the output wires corresponding to the goal type partition. This will create the following three graphs, paired with their corresponding proof state:



Note that the rightmost graph has terminated as the goal node is on the output wire.

Environment tactics are used to change the environments; this class of tactics is represented by the set *Env*, which in practise is any atomic tactic starting with "*ENV_*". These are in most cases theorem prover specific and evaluated by the *eval_env* function, which will return a set of new environments.

For a graph tactic (GRAPH), the environment is first constrained to the given arguments; then the child graph is evaluated by $\Downarrow^*_{GG}$. On termination, the output goals of the child are returned, once the environment has been updated as previously discussed. ADDINPUT is used to add an input goal to the graph. It follows the same approach as RIGHT by using the two combinators and the *gnds* function.

We state four key properties, without proof, which follows directly from the semantics:

Claim 1 *A goal can only be generated by a tactic application.*

Claim 2 *No goals are "lost", meaning that if a tactic generates a goal then it will appear on the graph and remain there until it has been discharged by a tactic.*

Claim 3 *No goals are duplicated, i.e. there is only a single instance of an open goal in the graph.*

Claim 4 *Evaluation will only change the* current *graph of a PSGraph.*

4 The Tinker tool[4]

The Tinker tool [17,30] implements PSGraph with support for the *Isabelle* [35], *ProofPower* [4] and *Rodin* [1] theorem provers. Tinker consists of two parts: the *CORE* and the *GUI*[5]. The rightmost shaded box of Fig. 7 is the CORE, while

[4] The Tinker source code is available from [18].

[5] The integration with Rodin [25] has a third part, which we will briefly outline in § 4.

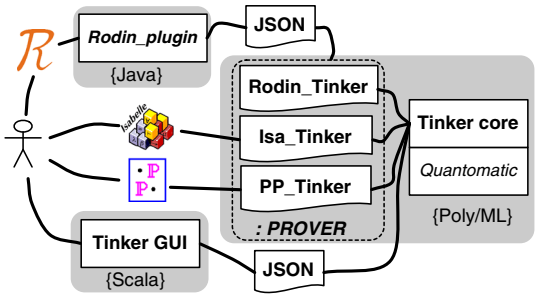


Fig. 7 Architecture of Tinker

the bottom-left box is the GUI. The CORE implements the main functions of Tinker, using the *Quantomatic* diagrammatic proof assistant to represent and rewrite graphs [23].

In § 5 we give details of the GUI, focusing on user features for working with the system. In the remainder of this section we will discuss the CORE. Details of how it is implemented and interacts with Quantomatic are described in [17]. Here, we will focus on how theorem proving independence is achieved and how the different provers have been integrated. For a more tutorial-like exposition of how to go about connecting a new prover, we refer to [28].

A theorem prover is connected by providing an implementation of an ML signature called `PROVER`. Figure 7 shows three implementations of this signature using three different ML structures: `Isa_Tinker` (Isabelle), `PP_Tinker` (ProofPower) and `Rodin_Tinker` (Rodin).

Proof states and goals The crucial part of a prover integration is the representation of proof states and goals. While Tinker uses the underlying prover's representation, these differ between the various provers. The user therefore has to implement the `PROVER` signature to represent these, possibly augmented with some additional bookkeeping information. These act as a bridge between Tinker and the theorem prover. The proof state must keep track of the goals that are "active" in the PSGraph, while from Tinker's point of view a goal is just a named element that can be used to interact with the proof state (e.g. to apply it to a tactic).

In Isabelle, a proof state is a theorem[6]. To prove a conjecture C, an initial proof state

$$C \Longrightarrow C$$

is created. Tactics are then applied to the antecedent of the implication. For example, a tactic that reduces C to the goals C_1 and C_2 will generate the theorem

$$C_1 \Longrightarrow C_2 \Longrightarrow C. \tag{1}$$

This process continues until all goals are proven and the user is left with the original conjecture C, for which there is now a proof. In Isabelle, a goal does not have a name; instead one provides the position of the goal when applying a tactic to it. For example, to apply a tactic to C_1 of (1), 1 is given as argument, while for C_2, 2 is given. In our proof state data structure we therefore augment the theorem with a map from names used in Tinker to the goal number they correspond to. A goal node will then use this name.

In ProofPower, the proof state is represented by an element of type `GOAL_STATE`. Here, each goal is named so we just use this name for the goal.

Rodin [1] is Eclipse-based and implemented in Java. As shown in Fig. 7, the Rodin integration consists of two components: a Tinker plugin for Rodin and an implementation of the `PROVER` signature by the `Rodin_Tinker` structure. These communicates over a JSON protocol. Here, the main functionality is in the plugin and the only thing done by the `Rodin_Tinker` structure is to handle communication between the Rodin plugin and Tinker CORE. In Rodin, the proof state is handled by the *Rodin Proof Obligation Manager (POM)*.

Executing a tactic In the formal semantics of Fig. 6, atomic tactics of the prover are evaluated by

$$eval(T, Args', P, gname)$$

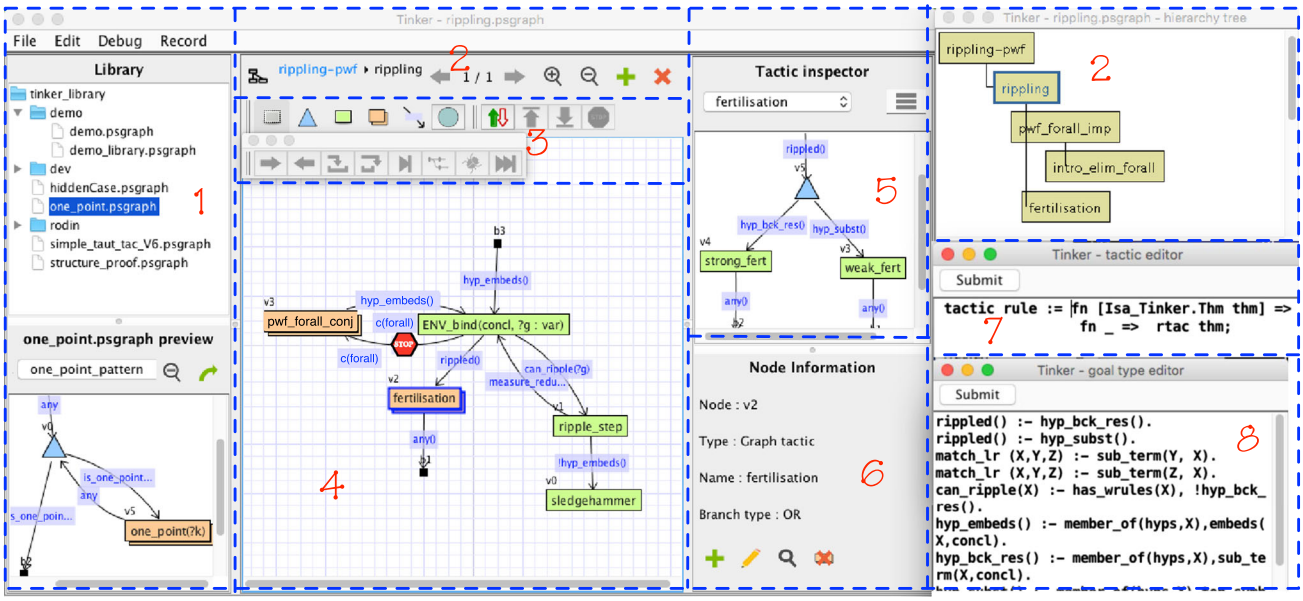
where T is the tactic name; $Args'$ are the arguments with all variables instantiated (by $[\![\text{-}]\!]_E$); P is the proof state; and *gname* is the name of the goal. This function should return a set (implemented as a lazy sequence) of pairs of a new proof state and a sequence of new goal names.

A key challenge when connecting a theorem prover to Tinker is that its internal representation of tactics has to be ported to work on the representation of proof states and goals from the `PROVER` signature.

Tactics without arguments (e.g. those in Fig. 3) can be handled by a generic "wrapper" function that turns the underlying prover representation to the one required by Tinker. In Isabelle, a tactic takes as argument the position of the goal[7]. "The wrapper" looks up the position from the name, then generates a new fresh name for each new goal and updates the map. ProofPower keeps track of the goals in a stack, and a tactic is applied to the goal that is on top of the stack. Thus, to apply a given tactic to a given goal, "the wrapper" first moves the goal to the top of the stack and then applies the tactic. In Rodin, the plugin is responsible for calling the correct tactic in the POM and sends the updated proof status back to the CORE.

[6] Isabelle has a sub-goal package, used, e.g. by Eisbach [31], which we plan to use in the future.

[7] Many Isabelle tactics also expect a proof context which we for simplicity ignore here.



1: Library panel
2: Hierarchy utilities
3: Drawing and evaluation controls
4: Graph panel
5: Hierarchical node inspector
6: Information panel
7:Tactic editor
8: Goal type editor

Fig. 8 Tinker GUI and its layout

If a tactic has arguments, then a user must manually implement a version of this tactic for Tinker where the arguments are represented as a list of a given type (called `arg_data`). This is a deep embedding of the supported types[8]. Except for this, it is handled the same way as a tactic without arguments.

Environment tactics Environment tactics only update the environment of a goal node. In the semantics of Fig. 6 the execution of an environment tactic is achieved by the *eval_env*$(T, Args, env)$ function. It will return a set of new environments. These are handled in a similar way as atomic tactics, albeit this is simpler as there is not a proof state to update.

5 Developing and debugging with the GUI[9]

The Tinker GUI provides users with an additional interface to support the development and debugging of proof strategies; for all other tasks the existing GUI of the underlying theorem prover is used.

5.1 Developing proof strategies

Figure 8 shows the Tinker GUI and its layout. Here, a user can draw a graph from the *Graph panel* by selecting the type of node from the *Drawing and evaluation controls* panel. Tactics are connected by dragging a line between them. When selecting an entity, the details are displayed in the *Information panel*, and they can be edited by double clicking[10]. It supports all the nodes of Fig. 2, albeit a user cannot draw a goal as this has to be created by the theorem prover.

Tinker allows "boxing" of sub-graphs into hierarchies, by a simple mouse click. Tinker also supports a range of features to work with hierarchies. In the *Hierarchical node inspector*, users can preview the internal structure of an hierarchical node. In the *Hierarchy utilities* panel (top right of Fig. 8), the hierarchical path of the current graph under edit is shown, as well as a tree view of the hierarchical structure of a PSGraph. It is also easy to move between and edit hierarchical nodes.

Graphical libraries Reuse of PSGraphs is supported by a *library*. This feature is provided in the *Library panel*. The items in the library are PSGraphs and stored in a special purpose folder. A new PSGraph can be added to the library by copying the relevant file to this directory. When importing an item from the library to the current PSGraph, Tinker will copy it to the graph that the user is currently editing and merge all the required information, such as defined tactics and goal types (see below).

[8] See [28] for a detailed example.

[9] Example screen casts can be found at [18].

[10] More details of running the tool is available from the user manual [8].

Defining new tactics Using the "wrapper" function described in § 4, a label of an atomic tactic is just treated as ML code and executed as a tactic in the CORE. Some tactics can be very long, or, e.g. use higher-order features that one would like to hide. While one can define shorthand notation of these in the prover, we have also developed a *tactic editor* in the GUI, which is shown in Fig. 8 (centre right ((7)). It expects the syntax

`tactic` ⟨*name*⟩ `:=` ⟨*ML code*⟩`;`

An atomic tactic box in the graph that is labelled by ⟨*name*⟩ will then apply ⟨*ML code*⟩. To illustrate, Isabelle supports higher-order resolution through a tactic called `rtac`, which in an Isabelle proof script (called Isabelle/Isar [39]) is written as `rule`. This tactic takes a theorem to resolve with as an argument. In the tactic editor, we can define this as:

```
tactic rule := fn [Isa_Tinker.Thm thm]
               => fn _ => rtac thm;
```

Note that: `fn` is ML syntax for lambda abstraction in Isabelle; `Isa_Tinker.Thm` is a constructor of the `arg_data` type mentioned in § 4; while the last argument (`fn _`) refers to the proof context of Isabelle, which can be ignored for this tactic.

Defining new goal types Tinker requires that a set of *atomic goal types* are provided by the underlying prover and made available to the CORE. The GUI provides a goal type editor, as shown in Fig. 8 (right), which implements the Prolog-inspired goal type language described in § 2.1. To illustrate, assume that the atomic goal type `top_symbol(S,T)`, discussed in § 2.1, is provided. We can then define `c(X)` in the editor as:

```
c(X) :- top_symbol(X,concl).
```

As is common in Prolog, we write "`:-`" for "←", and `conj` is Isabelle's ASCII representation of conjunction.

Recording and replaying Tinker provides features to export PSGraphs and record proofs. A PSGraph can be exported to the SVG format. The recording feature can be switched on/off to start/pause the recording of changes made to a graph. These changes could have been made by the user or by the tool during evaluation. Once completed, such recording can be exported to a lightweight web application (written using HTML, CSS and JavaScript) via a generated JSON file. Examples of recordings can be found at [18].

5.2 Debugging proof strategies

One can think of three modes of applying a PSGraph:

1. In an *automatic* (or normal) mode it is treated as a black box, and all you see is the resulting goals from applying it. This mode is the same as applying a normal tactic/method from the prover. In Isabelle/Isar, a PSGraph ⟨*psgraph*⟩ is executed by the command `apply tinker(`⟨`psgraph`⟩`)`; in ProofPower it is achieved by the command `apply_ps` ⟨*psgraph*⟩ while the  button is available for Rodin.
2. In an *interactive* mode, the steps of evaluation are inspected in the GUI, achieved by the commands `apply itinker` (Isabelle) and `apply_ps_i` (ProofPower).
3. In a *debugging* mode, which combines these modes as detailed below.

In the interactive mode, users can: (1) select which goal to apply; (2) choose between stepping into and stepping over the evaluation of graph tactics; (3) apply and complete the current graph tactic; (4) backtrack to the next branch in the search space; (5) apply and finish the whole proof strategy; (6) insert a breakpoint and evaluate a graph automatically until the breakpoint is reached by a goal. These options are illustrated in *Drawing and evaluations controls* panel of Fig. 8. The graph displayed there also shows a breakpoint.

Breakpoints A novel feature of Tinker is the support for breakpoints, which can be added/removed from wires by a simple mouse click. This was introduced in [28]. In presence of breakpoints in the debugging mode, we introduce a new definition of *termination*, with the difference from Definition 2 underlined:

Definition 8 (*Termination* (**debug mode**)) A graph has *terminated*, if for all goals g of the graph, the destination of the output wire of g is either the graph output, a breakpoint node, or another goal.

The semantics is updated to handle this mode by removing the rewrite rule for breakpoints from R (see Fig. 5).

In case of failure, the debugging mode can be used to identify and rectify errors that have caused evaluation to fail, and breakpoints are used to support this and act as an interface between the automatic and interactive mode. When debugging a large proof strategy (see [29]), one may have an idea of where the problem is and would like to avoid having to go through all steps until that point is reached. This can be achieved by adding a breakpoint node at the point where one would like to start stepping through the graph. When a graph terminates in debug mode, it stops instead of report-

ing success/failure. A user can then manually step through evaluation using the GUI features described above.

Logging To support debugging, an *evaluation log*, which shows the details of the current proof status, can be displayed. The log uses tags that can be used to filter the log to tags of interest. Tags include properties about: goals, goal types, tactics, environments (of goals) and failure information. An example of a log is

```
ENV_DATA : [k: E_Trm(Domain B.0)]
```

where `ENV_DATA` is a tag for the environment of the goal which is currently evaluated. We will see practical use of the logging mechanism in § 6.

6 Proof patterns as PSGraphs

To illustrate use of PSGraph and Tinker we encode three proof patterns of increasing complexity.

6.1 "Disjunctions to the top"

The first pattern we address is from a proof discussed in [15] of a heap case study in VDM [20, Chapter 7]. The problem is a limited form of normalising into a disjunctive normal form (DNF), where we would like to turn a goal of the shape

$$\exists x_1 \cdot \ldots \cdot \exists x_n \cdot (A \vee B) \wedge C$$

into the shape:

$$(\exists x_1 \cdot \ldots \cdot \exists x_n \cdot A \wedge C) \vee (\exists x_1 \cdot \ldots \cdot \exists x_n \cdot B \wedge C).$$

The strategy needs to first distribute the disjunction over a conjunction, then over all the existential quantifiers. Isabelle has two rewrite rules that can be applied to achieve this by rewriting. The first is *conj_disj_distribR*:

$$(P \vee Q) \wedge R \Leftrightarrow (P \wedge R) \vee (Q \wedge R)$$

It distributes a disjunction over a conjunction. The other is *ex_disj_distrib*:

$$\exists x \cdot P \vee Q \Leftrightarrow (\exists x \cdot P) \vee (\exists x \cdot Q)$$

which distributes a disjunction over an existential.

Figure 9 shows a PSGraph that implements a strategy to achieve this using these two theorems. It applies the Isabelle substitution tactic (*subst_tac*) in both cases. This tactic takes the rewrite rule to apply as an argument. It first distributes over the conjunction; then it repeatedly distributes over an existential until the disjunction is at the top, identified by

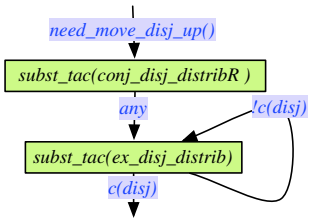


Fig. 9 Dealing with disjunctions

the *c(disj)* goal type; its negation labels the feedback loop to indicate that it should continue. A feature of the strategy (and PSGraph) is that it is explicit about the termination condition of the loop. The most interesting thing is to identify when this strategy is applicable, represented by the *need_move_disj_up*() goal type of the input wire:

$$\begin{aligned}
\mathit{need_move_disj_up}() &\leftarrow \mathit{into_all_ex}(\mathit{concl}, X),\\
&\quad\ \mathit{or_and}(X).\\
\mathit{into_all_ex}(X, Y) &\leftarrow \mathit{into_ex}(X, Z),\\
&\quad\ \mathit{into_all_ex}(Z, Y).\\
\mathit{into_all_ex}(X, X) &\leftarrow !\mathit{into_ex}(X, _).\\
\mathit{or_and}(X) &\leftarrow \mathit{match}(?A \wedge (?B \vee ?C), X).\\
\mathit{or_and}(X) &\leftarrow \mathit{match}((?A \vee ?B) \wedge ?C, X).
\end{aligned}$$

Here, $\mathit{into\_all\_ex}(X, Y)$ steps over all the existential binders of term X so that Y will have the body of the last binder; in our case it will be $(?A \vee ?B) \wedge ?C$. This uses the atomic goal type $\mathit{into\_ex}(X, Z)$ which takes a single step over an existential quantifier. It will fail if there is not an existential quantifier, which is the termination case for $\mathit{into\_all\_ex}(X, Y)$. It uses recursion and two atomic goal types to achieve this:

- $\mathit{match}(P, T)$ holds if term T matches pattern P. This uses Isabelle's pattern-matching capabilities for terms.
- $\mathit{into\_ex}(X, Z)$ steps into the body of an existential. For example, for $\mathit{into\_ex}(\exists x \cdot P, Z)$, Z will be bound to P.

This simple, yet common, proof strategy illustrates how to directly and intuitively represent a high-level proof pattern in PSGraph; in this case, a strategy which automates some standard proof steps, in order to "get to the meat of the proof". The strategy can easily be generalised, but for simplicity, the proof strategy has been tailored for a particular problem.

6.2 Existential quantifiers via the "one point rule"

Here, we develop a proof strategy for instantiating existentially bound variables through a technique known as the *one point rule*[15,40].

We will illustrate this through a type of proof obligation that is common in methods such as VDM [5,20] and Z [40] called *feasibility*. This is used to show that operations can be

executed, through the existence of a resulting "after" state. We will illustrate this with a feasibility proof obligations in Z for a case study of a telephone exchange system used in [27]:

$$\frac{s1 \neq s2 \qquad \forall x \cdot \forall y \cdot y \in call[\{x\}] \Rightarrow x \neq y}{\begin{array}{l}\exists callee' \cdot \exists call' \cdot \forall x \cdot \forall y \cdot y \in call'[\{x\}] \Rightarrow x \neq y \wedge \\ \quad callee' = \mathbf{dom}\, call' \wedge call' = call \cup \{(s1, s2)\}\end{array}}$$

Note that $R[X]$ is the image of R over the set X. The details are not important to follow the discussion; the key observation is that it starts with existential quantifiers and the bound variables appears alone in one side of an equality. For example, for $callee'$ it is $\mathbf{dom}\, call'$, meaning that we should, according to the one point rule, instantiate $callee'$ to $\mathbf{dom}\, call'$.

Version 0: debugging with Tinker The one point rule should first find the witness in the body of the quantifier and then instantiate the quantified variable with this witness. However, the quantifier in question may be prefixed by other existential variables (e.g. *call'* is prefixed by *callee'* in the above example). The overall proof strategy must therefore also be able to swap binders:

- first find the term to be used as a witness;
- then move the corresponding existential binder to the top, if it is not already at the top;
- finally, instantiate the binder with the witness.

Figure 10 shows an implementation of this technique in PSGraph. It has several new goal types:

- *is_one_point(X)* holds if the one point rule is applicable, i.e. X has the shape required;
- *is_top(K,T)* holds if term $x = K$ or term $K = x$ is a sub-term of T, and x is existentially bound at the top-level;
- *depth(K,T,D)* holds if term $x = K$ or term $K = x$ is a sub-term of T, and x has D preceding existential binders;
- *less(X,Y)* implements the order $X < Y$.

The proof strategy first finds the term of the conclusion to be used as a witness ?*k* by *ENV_one_point_match*. It then moves it to the top unless it is already at the top by *move_up*, before the witness is instantiated by *rule_tac(exI,?k)*.

The *move_up(?k)* graph tactic takes a single argument ?*k*, which means that the environment of a goal entering or leaving this box will only contain ?*k*. It has two new environment tactics:

- *ENV_one_point_match(T,V)* applies the "one point rule match" by finding the (first) instance of $x = K$ or $K = x$ in T, such that x is existentially bound (and only preceded by other existential binders). It will then bind term K to V in the goal type environment;
- *ENV_exists_depth(T,K,N)* will assign to N the number of binders preceding the binder of x in T, with $x = K$ or $K = x$.

The *move_up* graph tactic will first figure out the number of binders preceding the given binder ?*k* and bind this to ?*n*. It will then swap two existentials using the *ex_comm* rewrite rule:

$$\exists x \cdot \exists y \cdot P \Leftrightarrow \exists y \cdot \exists x \cdot P$$

There are three possible outcomes from applying this rewrite rule, where two are wanted. The first wanted outcome is that the binder is now on top. The goal will then exit the graph tactic and then apply the *rule(exI,?k)* tactic. The second wanted case is that we have successfully moved the binder one step up but we are not at the top. This is achieved by the *reduced(?k,?n)* goal type, whose schema is defined as

$$reduced(X, N) \leftarrow !is\_top(X, concl), depth(X, concl, D), less(D, N).$$

The first literal states that X (?*k*) is not at the top of *concl*; we then find the (new) depth D of X (?*k*) and make sure it is reduced compared with the old depth N (?*n*)[11]. The unwanted step is that we have either swapped two other binders or that we have swapped this binder the wrong way. This will not make any progress to the proof and is therefore discarded (as there is no output wire with a satisfying goal type).

When executing this strategy for our example, the proof strategy fails. We then use the interactive mode of Tinker and step through the proof to find the problem[12]. In this case it first finds the witness for *callee'*, and binds ?*k* to $\mathbf{dom}\, call'$. As $callee'$ is at the top-level it will enter the *rule_tac(exI,?k)* box and fail. We can then inspect the *Tinker logger*, which prints:

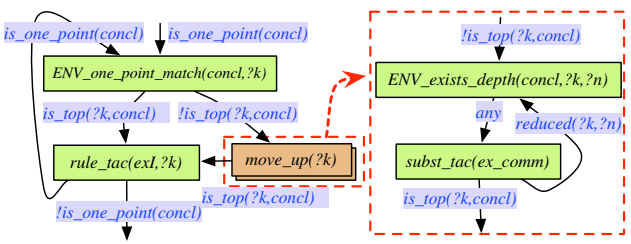


Fig. 10 One point rule in PSGraph (version 0)

[11] The idea of using such term difference is similar to the PRESS system for equational theories [38].

[12] If this was part of a larger tactic, then we could have used a breakpoint to start at the entry of this part of the strategy.

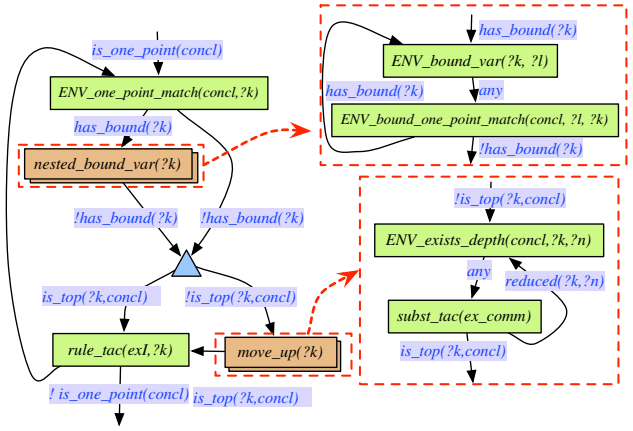


Fig. 11 One point rule in PSGraph (final version)

```
> FAILURE : Fail to apply tactic for pnode a
> ENV_DATA : [k: E_Trm(Domain B.0)]
```

From this we can see that the problem is to apply the underlying (atomic) tactic, given the argument for $?k$. The problem is indeed the argument: $?k$ is bound to $\mathbf{dom}\, call'$ but $call'$ itself is a bound variable with the binder missing (known as a "dangling variable").

A correct one point rule To overcome this type of problems we update the strategy as shown in Fig. 11. The goal type *has_bound*(X) checks if X contains any bound variables. If it does, then it enters the new sub-graph. There the first environment tactic, *ENV_bound_var*$(?k, ?l)$, will bind $?l$ to the bound variable within $?k$. For our example this is $call'$. The second environment tactic, *ENV_bound_one_point_match*$(concl, ?k, ?l)$, works in a similar way to *ENV_one_point_match*. It will find the term that $?l$ equals (using the "one point rule match") and map $?k$ to this term. In our case, $?k$ will be mapped to $call \cup \{(s1, s2)\}$ as a result. As $?k$ may contain another bound variable this process will iterate[13]. In our case it will only make a single iteration and the variable will be instantiated. After two applications of the overall strategy, the conclusion of the goal becomes:

$$\forall x \cdot \forall y \cdot y \in (call \cup \{(s1, s2)\})[\{x\}] \Rightarrow x \neq y \wedge \\ \mathbf{dom}\,(call \cup \{(s1, s2)\}) = \mathbf{dom}\,(call \cup \{(s1, s2)\}) \wedge \\ call \cup \{(s1, s2)\} = call \cup \{(s1, s2)\}$$

6.3 Rippling

The last two conjuncts of the conclusion follow from reflexivity, which gives the goal:

$$\frac{s1 \neq s2 \qquad \forall x \cdot \forall y \cdot y \in call[\{x\}] \Rightarrow x \neq y}{\forall x \cdot \forall y \cdot y \in (call \cup \{(s1, s2)\})[\{x\}] \Rightarrow x \neq y}$$

[13] We assume absence of circularity.

One can see that the conclusion is the same as one of the hypotheses, but with additional "stuff". Formally, there is an *embedding* of this hypothesis into the conclusion [10]. A standard strategy for this type of goal is to rewrite the conclusion towards this hypothesis. This strategy is called *rippling* [10]. The second author's Ph.D. thesis addressed rippling for similar types of proof obligations [27].

Figure 12 shows such an encoding of rippling in PSGraph. It is based upon [27] and extended with a limited form of a technique called *piecewise fertilisation* [3] (graph tactic *pwf_forall_conj*).

The *hyp_embeds* goal type expresses the "embedding" property and is a requirement to start rippling:

$$hyp\_embeds() \leftarrow member(hyps, X), embeds(X, concl).$$

Next, *ENV_bind* binds the conclusion to variable $?g$. This box has three output wires. One of them is labelled by *can_ripple(?g)*, which means rippling can be started:

$$can\_ripple(X) \leftarrow has\_wrules(X), !hyp\_bck\_res(). \\ hyp\_bck\_res() \leftarrow member(hyps, X), sub\_term(X, concl).$$

Note that we assume the existence of a set of valid rewrite rules, which are called *wave rules* in rippling. *has_wrules(X)* is an atomic goal type which checks that there is an applicable wave rule to apply to X. The second clause, $!hyp\_bck\_res()$, checks that backward resolution with a hypothesis is not applicable. A key feature of rippling is that it guarantees termination. This is achieved by the *measure_reduces(?g)* goal type, which ensures that the current conclusion has reduced this measure compared with $?g$ (the conclusion before the step was made). We refer to [10] for more details of this measure. Any non-rippling goal ($!hyp\_embeds()$) is sent to Isabelle's powerful *sledgehammer* tool [36]. After a few ripple steps, our goal becomes:

$$\frac{s1 \neq s2 \qquad \forall x \cdot \forall y \cdot y \in call[\{x\}] \Rightarrow x \neq y}{\forall x \cdot \forall y \cdot y \in call[\{x\}] \Rightarrow x \neq y \wedge \\ y \in \{(s1, s2)\}[\{x\}] \Rightarrow x \neq y}$$

Logical connectives and quantifiers are handled by the *piecewise fertilisation* graph tactic *pwf_forall_conj*. It will skolemise the universal quantifiers in the goal and instantiate the quantifiers in the hypothesis with the newly introduced skolem functions (*intro_elim_forall* graph tactic). It then breaks up the conjunction in the conclusion. This is the second output wire of *ENV_bind* and identified by the *c(forall)* goal type. This property holds for our goal. Piecewise fertilisation will generate two goals, where for the first there is no embedding and it sent to *sledgehammer* that discharges it. The second goal is:

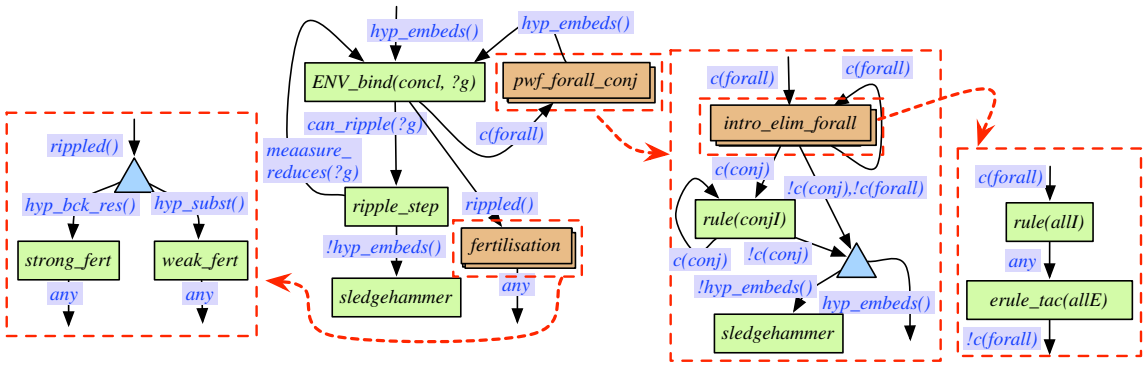


Fig. 12 A rippling-based strategy

$$\frac{s1 \neq s2 \qquad y \in call[\{x\}] \Rightarrow x \neq y}{y \in call[\{x\}] \Rightarrow x \neq y}$$

The third and final output of *ENV_bind* is labelled by the goal type:

$$rippled() \leftarrow hyp\_bck\_res().$$
$$rippled() \leftarrow hyp\_subst().$$

This means that the goal is fully rippled. The goal then enters the final step of rippling, which is called *fertilisation*. This is encoded in the *fertilisation* graph tactic of Fig. 12. Here, the hypothesis that was embedded in the conclusion is applied, and there are two cases. The first case, identified by the *hyp_bck_res*() goal type, is called *strong fertilisation* and applies the hypothesis as backward resolution. The other case, identified by the *hyp_subst*() goal type, is called *weak fertilisation* and will rewrite the conclusion with the hypothesis. The *strong_fert* and *weak_fert* atomic tactics are minor extensions to Isabelle's backward resolution and substitution tactics, respectively. These extensions ensure that the correct hypothesis is applied.

In our case, strong fertilisation will discharge the goal. This concludes rippling and illustrates how to represent a complex and well-known proof technique in PSGraph. We believe that the graphical view helps in explaining how this technique works.

7 Related work

This paper extends and builds on [30], where we introduced new features of the Tinker GUI. We also build on [16,17], where the Tinker tool and PSGraph were first introduced. In [28] we develop the goal types and environment tactics used here, as well as the breakpoints and logging features—with the formal semantics for evaluation with breakpoints and environments tactics given here. [28] also contains an empirical study of PSGraph and Tinker via a set of re-engineered case studies developed in ProofPower. This can be contrasted to this paper, where we show how to implement existing proof patterns from scratch in PSGraph. In [29], we show industrial application of PSGraph with ongoing work of implementing D-RisQ Software Systems' (www.drisq.com) highly complex Supertac tactic in ProofPower [32]. Several of the features described here have been motivated by this work. We have also previously developed a Rodin version of Tinker [25], which we have briefly described in Sect. 4.

Tinker is built on top of the Quantomatic graph-rewriting engine [23], which is used internally as a library function. There is also a web-based version of Tinker, which supports a subset of the GUI features discussed here [7]. There has been a considerable amount of work on visualising *proof trees*, including: LUI [37] for mega; XIsabelle [33] for Isabelle; ProveEasy [11] and Jape [6] for teaching; and some more recent work for Mizar [26,34]. However, none of these visualise the high-level strategy. Moreover, in PSGraph the diagram is not just a visualisation of the proof strategy—it is the proof strategy! Bundy [9] has argued for the role of *explanation* for proof strategies, and we hope that we have shown how PSGraph can help explaining the strategy of a proof in addition to be used to guide the search. Such explanation is important for maintenance when team changes, and our work with D-RisQ [29] has had very promising initial results when porting their proof tactics from Ada to C verification.

While there are tactic languages that support robust and user-friendly tactics (e.g. Ltac [12] for Coq and Eisbach [31] for Isabelle), we believe that the development and debugging features of Tinker are novel. The most relevant tool that we are familiar with to support debugging tactics is the *Tactician* tool for HOL light [2]. In HOL light, a proof can be a sequence of (interactive) "apply" steps, or they can be combined into a single step (by means of tactic combinators) which is then applied. Tactician is a tool to fold sequences into a single tactic and unfold a tactic into a sequence of steps. This can then be used for debugging by enable users

to step through a large tactic, similar to how this can be achieved with Tinker. However, it only works for a small subset of ML and it is not clear how this approach can be generalised to arbitrary tactics. Moreover, it unfolds only one particular branch of the proof which does not necessarily reflect the underlying proof strategy. Another tool recently developed to support debugging is a reasonably new tracing mechanism for the *simp* tactic in Isabelle [19]. This is implemented as a plugin for the Isabelle/jEdit Prover IDE. It supports hierarchical viewing of simplification traces, and, as with Tinker, it enables breakpoints to be inserted where the user can step through and interact with the tactic. The breakpoints can either be an application of certain theorems or if the goal matches certain patterns. Note that it is not used to debug the (sub)tactics used to implement the simplifier: it will only show how the simplifier applies rewrite steps. Our logging mechanism is considerably simpler and closer to the more rudimentary ones supported in other ITP systems (including the previous tracing mechanism for the *simp* tactic in Isabelle). However, in practice we have found that our logging mechanism is sufficient as it only relates to a step at a time, while the *simp* tactic could involve hundreds of steps.

Rippling has been implemented in several systems, the closest being *IsaPlanner* [13]. In his Ph.D. thesis, Lin addressed rippling for Event-B invariant proofs [27], which is comparable to the Z representation used here. This was implemented in IsaPlanner, which has a similar composition language to tacticals that we used to motivate PSGraph in § 1. Finally, in [16] we developed a very basic and ad hoc PSGraph version of rippling, which we have considerably improved upon and extended here. One key difference is the more sophisticated goal type language, which enables us to make more details of rippling declarative and available to the users (as opposed to "hidden" in the boxes).

8 Conclusion and future work

We have given a detailed and formal account of the semantics of PSGraph and shown how this is implemented in Tinker, with support for multiple interactive theorem provers. We have shown how to develop and debug proof strategies using the GUI of Tinker and illustrated use of the language and tool by encoding several existing proof strategies from scratch in PSGraph.

In the future, we would like to improve static checking of PSGraph, such as being able to validate a PSGraph before evaluation. We also plan to improve the layout algorithm and develop and implement a better framework for combining evaluation and user edits of PSGraphs. We are also working on a more lightweight version of Tinker (independent of the underlying Quantomatic tool), which will make it easier to install and maintain. There have been recent advances in the expressiveness of *pattern graphs* [22,23], which will enable us to simplify the graphical parts of the evaluation. We are also working on improving features for parametrised graph tactics, in order to improve reuse.

Acknowledgements An initial version of the one point rule in PSGraph was developed with Iain Whiteside. Thanks to Pierre Le Bras, who implemented the Tinker GUI [30], Aleks Kissinger, Rob Arthan, Colin O'Halloran and members of the AI4FM project for valuable discussions.

References

1. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J.Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
2. Adams, M.: Refactoring proofs with tactician. In: Bianculli, D., Calinescu, R., Rumpe, B. (eds.) Software Engineering and Formal Methods: SEFM 2015, Revised Selected Papers, pp. 53–67. Springer, Berlin (2015)
3. Armando, A., Smaill, A., Green, I.: Automatic synthesis of recursive programs: the proof-planning paradigm. Autom. Softw. Eng. **6**(4), 329–356 (1999)
4. Arthan, R., Jones, R.B.: Z in HOL in ProofPower. BCS FACS FACTS (2005-1). http://www.bcs.org/upload/pdf/facts200503.pdf
5. Bjørner, D., Jones, C.B. (eds.): The vienna development method: the meta-language. Lecture Notes in Computer Science, vol. 61. Springer (1978). doi:10.1007/3-540-08766-4
6. Bornat, R., Sufrin, B.: Jape: A calculator for animating proof-on-paper. In: McCune, W. (ed.) CADE-14, pp. 412–415. Springer, Berlin (1997). doi:10.1007/3-540-63104-6_41
7. Bras, P.L.: Web based interface for graphical proof strategies (2015). Undergraduate CS Honours Thesis. https://goo.gl/LWG522
8. Bras, P.L., Grov, G., Lin, Y.: Tinker: user guide. http://ggrov.github.io/tinker/userGuides.pdf
9. Bundy, A.: A science of reasoning. In: de Swart, H. (ed.) International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, pp. 10–17. Springer, Berlin (1998)
10. Bundy, A., Basin, D., Hutter, D., Ireland, A.: Rippling: Meta-level Guidance for Mathematical Reasoning, Cambridge Tracts in Theoretical Computer Science, vol. 56. Cambridge University Press, Cambridge (2005)
11. Burstall, R.: Proveeasy: Helping people learn to do proofs. ENTCS **31**, 16–32 (2000). doi:10.1016/S1571-0661(05)80327-5
12. Delahaye, D.: A Proof Dedicated Meta-Language. Electronic Notes in Theoretical Computer Science **70**(2), 96–109 (2002)
13. Dixon, L., Fleuriot, J.: Higher Order Rippling in IsaPlanner. In: Slind, K., Bunker, A., Gopalakrishnan, G. (eds.) TPHOL, pp. 83–98. Springer, Berlin (2004)
14. Dixon, L., Kissinger, A.: Open graphs and monoidal theories. Math. Struct. Comput. Sci. 23(2), 308–359 (2013). doi:10.1017/S0960129512000138

15. Freitas, L., Whiteside, I.: proof patterns for formal methods. In: International Symposium on Formal Methods, pp. 279–295. Springer, Berlin (2014)
16. Grov, G., Kissinger, A., Lin, Y.: A graphical language for proof strategies. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR, pp. 324–339. Springer, Berlin (2013)
17. Grov, G., Kissinger, A., Lin, Y.: Tinker, tailor, solver, proof. In: Benzmüller, C., Paleo, B.W. (eds.) UITP 2014, EPTCS, vol. 167, pp. 23–34. Open Publishing Association, London (2014)
18. Grov, G., Lin, Y.: Tinker webpage—resources for STTT paper. http://ggrov.github.io/tinker/sttt16/. Accessed Feb 2017
19. Hupel, L.: Interactive simplifier tracing and debugging in Isabelle. In: Watt, S.M., Davenport, J.H., Sexton, A.P., Sojka, P., Urban, J. (eds.) CICM, pp. 328–343. Springer, Berlin (2014)
20. Jones, C.B., Shaw, R.C.: Case Studies in Systematic Software Development. Prentice Hall, Upper Saddle River (1990)
21. Kissinger, A., Merry, A., Soloviev, M.: Pattern graph rewrite systems. CoRR arXiv:1204.6695 (2012)
22. Kissinger, A., Zamdzhiev, V.: Equational reasoning with context-free families of string diagrams. In: Parisi-Presicce, F., Westfechtel, B. (eds.) Graph Transformation, pp. 33–47. Springer, Berlin (2015)
23. Kissinger, A., Zamdzhiev, V.: Quantomatic: a proof assistant for diagrammatic reasoning. In: Felty, A.P., Middeldorp, A. (eds.) CADE-25, LNCS, vol. 9195, pp. 326–336. Springer, Berlin (2015)
24. Larkin, J.H., Simon, H.A.: Why a diagram is (sometimes) worth ten thousand words. Cognit. Sci. **11**(1), 65–100 (1987)
25. Liang, Y., Lin, Y., Grov, G.: The Tinker for rodin. In: Butler, M., Schewe, K.D., Mashkoor, A., Biro, M. (eds.) 5th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, pp. 262–268. Springer, Berlin (2016)
26. Libal, T., Riener, M., Rukhaia, M.: Advanced proof viewing in ProofTool. In: Benzmüller , C., Woltzenlogel, B. (eds.) UITP 2014, EPTCS, vol. 167, pp. 35–47. Open Publishing Association, London (2014)
27. Lin, Y.: The use of rippling to automate event-B invariant preservation proofs. Ph.D. thesis, The University of Edinburgh (2015)
28. Lin, Y., Grov, G., Arthan, R.: Understanding and maintaining tactics graphically OR how we are learning that a diagram can be worth more than 10K LoC. J. Formaliz. Reason. **9**(2), 69–130 (2016)
29. Lin, Y., Grov, G., O'Halloran, C.: G., P.: A Super Industrial Application of PSGraph. Butler, M.J., Schewe, KD., Mashkoor, A., Biró M. (eds.) 5th International Conference on Abstract State Machines. Alloy, B, TLA, VDM, and Z, pp. 319–325. Springer, Berlin (2016)
30. Lin, Y., Le Bras, P., Grov, G.: Developing and debugging proof strategies by Tinkering. In: Chechik, M., Raskin, J.F. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 573–579. Springer, Berlin (2016)
31. Matichuk, D., Wenzel, M., Murray, T.: An Isabelle proof method language. In: Klein, G., Gamboa, R. (eds.) 5th International Conference on Interactive Theorem Proving, pp. 390–405. Springer, Cham (2014)
32. O'Halloran, C.: Automated verification of code automatically generated from simulink. ASE **20**(2), 237–264 (2013)
33. Ozols, M.A., Cant, A., Eastaughffe, K.A.: Xisabelle: A system description. In: McCune, W. (ed.) CADE-14, pp. 400–403. Springer, Berlin (1997)
34. Pak, K.: The algorithms for improving and reorganizing natural deduction proofs. Stud. Logic Gramm. Rhetor. **22**(35), 95–112 (2010)
35. Paulson, L.C.: Isabelle: A Generic Theorem Prover, vol. 828. Springer, Berlin (1994)
36. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. IWIL-2010 **1**, 1–11 (2010)
37. Siekmann, J.H., Hess, S.M., Benzmüller, C., Cheikhrouhou, L., Fiedler, A., Horacek, H., Kohlhase, M., Konrad, K., Meier, A., Melis, E., Pollet, M., Sorge, V.: LOUI: lovely OMEGA user interface. Form. Asp. Comput. **11**(3), 326–342 (1999)
38. Sterling, L., Bundy, A., Byrd, L., O'Keefe, R., Silver, B.: Solving symbolic equations with PRESS. In: Calmet, J. (ed.) Computer Algebra, no. 144 in LNCS, pp. 109–116. Springer, Berlin (1982). Also available in J. Symbol. Comput. **7**, pp 71–84 (1989)
39. Wenzel, M.: Isabelle/Isar – a versatile environment for human-readable formal proof documents. Ph.D. thesis, Technische Universität München (2002)
40. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof, vol. 39. Prentice Hall, Upper Saddle River (1996)